

3D-SPIELEENTWICKLUNG MIT JAVA

Seminararbeit von
JULIAN WADEPHUL UND FLORIAN ROTTACH

an der KIT-Fakultät für Wirtschaftswissenschaften

eingereicht am : 26. Januar 2017
Studiengang : Wirtschaftsingenieurwesen

Institut für Angewandte Informatik und Formale Beschreibungsverfahren
KIT - Die Forschungsuniversität in der Helmholtz-Gemeinschaft

ZUSAMMENFASSUNG

In dieser Seminararbeit wird die 3D-Spieleprogrammierung in Java behandelt. Zur Vereinfachung des Prozesses wird eine Game - Engine namens jMonkeyEngine3 verwendet. Die inkludierte Entwicklungsumgebung stellt die Grundfunktionen von Spielen bereit und ermöglicht, dass sich der Entwickler auf das Spiel selbst konzentrieren kann. Auf dieser Basis wurde ein kleines Spiel programmiert, anhand welchem die fundamentalen Ideen und Umsetzungen von 3D-Programmierung geschildert werden. Dabei wird detailliert auf spezielle Anwendungen und wichtige Grundvorgehensweisen eingegangen, welche in allen heutigen 3D Spielen notwendig sind.

INHALTSVERZEICHNIS

1 EINLEITUNG	1
2 3D SPIELEENTWICKLUNG MIT JAVA	2
2.1 Allgemeiner Aufbau von 3D Spielen	2
2.2 Funktionsweise und Auswahl von Game Engines	3
2.2.1 Die jMonkeyEngine	3
2.3 Umsetzung in Programmcode	4
2.3.1 Erzeugung der Application-Klasse: SimpleApplication	4
2.3.2 Funktionsweise von Nodes	4
2.3.3 Modelle und Assets	5
2.3.4 Materialien	6
2.3.5 User-Input	6
2.3.6 Kollisionserkennung	7
2.3.7 Erzeugung einer Spielumgebung	7
2.3.8 Hinzufügen von Audio	11
2.3.9 Physikalische Modellierung	12
2.3.10 Effekte und Details	12
2.4 Optimierung des Programms	14
2.5 Kritik an der jMonkeyEngine3 und Fazit	17
LITERATURVERZEICHNIS	18

ABBILDUNGSVERZEICHNIS

Abbildung 1	Beispiel eines 3D Spiels	2
Abbildung 2	Materials auf Modellen	6
Abbildung 3	New Empty jME3 Scene mit erstelltem Terrain	7
Abbildung 4	Terrain mit Skybox	8
Abbildung 5	PointLight	9
Abbildung 6	DirectionalLight	10
Abbildung 7	SpotLight	11
Abbildung 8	Exponentieller Nebel - Ergebnis	12
Abbildung 9	Partikeleffekt - Ergebnis	13
Abbildung 10	Die jMonkey Figur in verschiedenen LODs	15
Abbildung 11	LODs mit der SDK im SceneExplorer erstellen	15

TABELLENVERZEICHNIS

Tabelle 1	Engines	3
-----------	-------------------	---

EINLEITUNG

Im Rahmen des Seminars „Programmieren 3: Betriebliche Informationssysteme soll in dieser Seminararbeit die Thematik der 3D-Spieleentwicklung vorgestellt werden. Das Projekt soll jedoch nicht nur in einer schriftlichen Ausarbeitung, sondern auch in einem Workshop mit den Seminarteilnehmern behandelt werden. In dem Workshop gilt es die grundsätzlichen theoretischen Hintergrundinformationen darzustellen, und in einem praktischen Teil anhand von Aufgaben selbst etwas zu programmieren. Um den Seminarteilnehmern einige Inhalte zu vermitteln, haben wir uns entschlossen, ein eigenes 3D-Spiel zu programmieren, welches im Workshop vervollständigt werden soll. Neben der veranschaulichten Darstellung der zu lernenden Inhalte, zeigt unser eigenes 3D-Spiel auf, was mit Java in der 3D-Spieleprogrammierung möglich ist. In dieser Seminararbeit werden wir die Grundlagen der 3D-Spiele-Programmierung behandeln, und dies anhand unseres 3D-Spiels veranschaulichen. Im folgenden möchten wir unser Spiel kurz vorstellen.

Unser entwickeltes Spiel 'Progman' ist eine Anlehnung an das bekannte Horror-Spiel 'Slenderman'. In unserer modifizierten Version geht es darum, dass der Spieler sich in einem Wald befindet und dort die 9 Bücher über anderen Themen in dem Seminar finden muss. Dabei wird er jedoch von einer Figur verfolgt, dem Progman, welcher versucht den Spielenden zu fangen. Ganz wesentlich hierbei ist die gruselige Stimmung, die durch Licht, Sound und Modelle in der Welt generiert wird. Im Laufe des Spiels nähert sich der Progman immer mehr an, bis er den Spieler gefunden hat. Dabei gehen Faktoren wie die Anzahl der bereits eingesammelten Bücher und die häufige Benutzung der Taschenlampe in die Geschwindigkeit des Progman mit ein. Gewonnen hat der Spieler, wenn er alle 9 Bücher eingesammelt hat, bevor er vom Progman gefangen wurde.

Für die Programmierung eines 3D-Spiels in Java stehen verschiedene Engines zur Verfügung. Diese Engines beinhalten vorgefertigte Klassen und Methoden, welche das Programmieren eines 3D-Spiels deutlich vereinfachen. Wir haben uns für die jMonkeyEngine entschieden, auf die Gründe möchten wir später noch genauer eingehen. Außerdem werden wir verschiedene Themen der Umsetzung im Programmcode und der jMonkey SDK untersuchen. Am Ende möchten wir noch die Optimierung eines 3D-Spiels beschreiben, da ein nicht-optimiertes 3D-Spiel sehr schnell zu aufwendig werden kann.

3D SPIELEENTWICKLUNG MIT JAVA

2.1 ALLGEMEINER AUFBAU VON 3D SPIELEN

Bei 3-dimensionalen Spielen wird das Spielgeschehen in einen Raum transferiert und dem Spieler die Möglichkeit gegeben sich dort frei bewegen zu können.

Im Gegensatz zu 2-dimensionalen Spielen sind hier deutlich mehr Berechnungen auf vektorieller Ebene notwendig, was sich auf die Laufzeit niederschlägt. Daher ist es besonders wichtig ein effizientes Programm zu generieren. Diesbezüglich ist Java nicht die optimale Programmiersprache, da durch den Interpreter viel Zeit verloren geht.

Dennoch können mit Java sehr schöne und funktionale Spiele erstellt werden.

Im Allgemeinen kann man sagen, dass die meisten 3D Spiele folgende Elemente enthalten:

- 3D-Modelle und eine räumliche Spielumgebung
- Eine sog. "Kamera", welche nur den relevanten Teil des Bildes abbildet
- Möglichkeiten der Interaktion
- Soundeffekte und Musik



Abbildung 1: Ausschnitt eines 3D Spiels in Java von ThinMatrix [Th]

2.2 FUNKTIONSWEISE UND AUSWAHL VON GAME ENGINES

Um nicht sämtliche mathematischen Berechnungen auf der Grafikkarte selber programmieren, oder beispielsweise die Lautsprecher für Audio-Effekte ansprechen zu müssen, erhält der Entwickler Unterstützung durch sogenannte "Game Engines". Diese beinhalten die Basisfunktionen von Spielen und ermöglichen dem Spiele-Programmierer eine gezieltere Entwicklung. Zu den Basisfunktionalitäten gehören im Allgemeinen die folgenden [Ba]:

1. Grafik-Engine
2. Physiksystem
3. Soundsystem
4. Zustandsspeicherung
5. Steuerung
6. Datenverwaltung

Zur Auswahl stehen eine Vielzahl von verschiedenen Engines, welche jeweils Vor- und Nachteile mit sich bringen. Da wir auf jeden Fall lernen wollten, wie die grundlegenden Dinge funktionieren, haben wir nach Engines gesucht, welche nur die Basisfunktionalitäten unterstützen, jedoch keine automatische Codegenerierung, Drag and Drop oder Editoren beinhalten. Im folgenden eine Übersicht einiger Engines:

GAMEENGINE	VORTEILE	NACHTEILE
Unity	Viele Benutzer, beliebt	Zu oberflächlich, kein Java
jMonkeyEngine	Sehr entwicklungsnahe, Java	Schlechte Dokumentation
Wurfel Engine	Benutzerfreundlich	Keine Physikunterstützung
Cry Engine	Sehr schöne Grafik	Kein Java

Tabelle 1: Vor - und Nachteile einiger Game Engines [Ue]

Letztendlich fiel die Entscheidung auf die jMonkeyEngine, welche häufig von Java Entwicklern verwendet wird.

2.2.1 Die jMonkeyEngine

Die jMonkeyEngine (jME) ist komplett in Java geschrieben und basiert auf dem Buch "3D Game Engine Design" von David Eberly [Eb]. Durch eine Abstraktionsschicht kann jedes beliebige Rendering System verwendet werden, beispielsweise die Lightweight Java Game Library (LWJGL) oder die Open Graphics Library (OpenGL). Die neueste Version ist jME3, welche einige hilfreiche Funktionen mit sich bringt, wie beispielsweise ein Partikelsystem, Frustum Culling oder 3D Sound Unterstützung [jMa].

FRUSTUM CULLING: "Frustum Culling ist eine Optimierungsmethode, bei der alle Objekte vom Zeichnen ausgeschlossen werden, die außerhalb des Sichtbereichs (des Frustums) liegen." [De]

2.3 UMSETZUNG IN PROGRAMMCODE

Im folgenden wird beschrieben wie einzelne Elemente in der jMonkeyEngine programmiert werden können und was dabei zu beachten ist. Die Erklärungen orientieren sich hierbei an dem jme3 Online-Beginners-Guide [jMb] und der entsprechenden Dokumentation.

2.3.1 Erzeugung der Application-Klasse: SimpleApplication

Die Main-Klasse jedes jME3 Spiels erbt von der Klasse SimpleApplication, welche ein Spiel darstellt. In der main-Methode wird dann eine neue Instanz erstellt und anschließend gestartet.

Jede Unterklasse der SimpleApplication beinhaltet die folgenden Methoden:

1. simpleInitApp(): Sorgt für das Laden von Modellen, der Erstellung einer räumlichen Umgebung sowie jegliche Initiierungen.
2. simpleUpdate(float tpf): Wird für jedes frame per second (fps) ausgeführt und kümmert sich um gegebenenfalls geänderte Spielzustände.
3. simpleRender(RenderManager rm): Wird stets nach simpleUpdate aufgerufen und zeichnet das Sichtbild des Spielers neu. Dazu bekommt die Methode einen RenderManager übergeben, welcher Präferenzen beim Zeichnen berücksichtigt (z.B. welche Ebene vorne oder hinten gezeichnet werden soll).

Die erste der drei Methoden wird stets zu Beginn ausgeführt um alle benötigten Elemente bereit zu stellen.

2.3.2 Funktionsweise von Nodes

Um Elemente zum Renderingprozess hinzuzufügen, um sie also sichtbar zu machen, müssen diese an entsprechende "Nodes"(engl. Knoten) angehängt werden. Hierbei gibt es je nach Verwendungszweck verschiedene Arten zum Beispiel die *audioNode* für Soundobjekte oder die *guiNode* für Elemente auf der Benutzeroberfläche.

Final werden alle Nodes an die rootNode, also die Wurzel, angehängt. Im Programmcode funktioniert dies mit der Methode *attachChild()* bzw. *detachChild()* zum entfernen.

Selbstverständlich können Objekte auch direkt an die rootNode angehängt werden, weshalb die Methodenparameter Modelle, Nodes, Bilder aber auch beispielsweise Audio-Files sind. Allerdings ist es empfehlenswert eine geeignete Baumstruktur zu erstellen um so bestimmte Elemente in Gruppen anzusprechen.

Beispiel: Erzeugung einer eigenen Node durch:

```
Node myNode = new Node();
myNode.doSomething();
```

Wird nun beispielsweise die Funktion *doSomething()* auf dem Konten ausgeführt, so wird diese auch für sämtliche Kinder des Knotens ausgeführt.

2.3.3 Modelle und Assets

Alle externen Gegenstände des Spiels werden im *assets*-Ordner im jME3 Projekt gesammelt. Dies sind multi-media Dateien wie 3D-Modelle, Soundfiles, Texturen, Shaders und was sonst noch benötigt wird. Um diese aus dem Ordner ins Spiel zu laden wird der sogenannte *AssetManager* benötigt, welcher einfach eine Instanz der Klasse mit entsprechenden Funktionalitäten ist.

Modelle sind dreidimensionale Gebilde welche verschiedenste Elemente in einem Spiel sein können. Hierbei verwendet jME3 die Klasse Spatial (engl. für "räumlich"). Zum Laden eines Objektes wird die entsprechende Funktion *loadTexture(String path)* bzw. *loadModel(String path)* aufgerufen und der entsprechende Pfad zum Modell übergeben:

```
Spatial baum = assetManager.loadTexture("Models/Baum.j3o");
```

Für Modelle gibt es viele verschiedene Datentypen. Neben dem jMonkey-eigenen Dateiformat .j3o existieren einige weitere. Selbstverständlich ist meist eine Konversion zwischen den Formaten möglich. Die häufigsten von uns angetroffenen Vertreter für Modell-Deklarationen sind die folgenden:

1. XML-Dateien: Aus einer mesh.xml Datei wird ein Objekt erzeugt.
2. OBJ-Dateien: Ein von *Wavefront Technologies* entwickeltes Dateiformat für geometrische Formen. [OB]
3. Blender-Dateien: Dies sind Dateien aus der Blender-Software, mit welcher Modelle erzeugt werden können.

Wie bereits unter *Funktionsweise von Nodes* beschrieben müssen die Spatials nun lediglich zur rootNode, bzw. einer anderen Node welche mit der rootNode verknüpft ist, hinzugefügt werden. Damit werden die Modelle und Texturen sichtbar und sind Teil des Rendering-Prozesses.

```
rootNode.attachChild(baum);
```

Der Aufbau von Modellen erfolgt durch entsprechende Software mit Polygonzügen oder Punktwolken. Dies definiert die allgemeine Struktur von Objekten.

Neben dieser und dem Material (vgl. *Materialien*) gibt es noch das Skelett. Dieses kann ebenfalls in einem entsprechenden Programm wie Blender erzeugt und daraus bestimmte Bewegungsabläufe in Spielen bestimmt werden. Das Skelett ist notwendig für Animationen von Modellen wie beispielsweise Gehen, Springen oder Ähnlichem. In unserem Progman-Spiel haben wir uns für eine First-Person Perspektive entschieden, wodurch keine Animationen für den Spieler notwendig waren. Darüber hinaus haben wir uns auf Grund des Zeitaufwandes gegen eine Implementierung eines Skeletts beim Progman entschieden. Dieser ist daher nur eine starre Figur.

2.3.4 Materialien

Da in Modell-Software vorerst nur die Form von Figuren bestimmt wird, muss anschließend noch die Farbe, die Oberflächenstruktur und das Lichtverhalten bestimmt werden. Dies funktioniert über sogenannte '*materials*'.

Die Datei für das Material kann in jme3 mit der Endung .mtl identifiziert werden. Diese bildet die entsprechenden Farbwerte von Bilddateien auf das Modell ab. Als Farbgeber sind beispielsweise mat.jpg oder mat.png gängig. Darüber hinaus können durch eine *bump-map* und *normalen*-Formate die Struktur der Oberfläche sowie das Verhalten bei Lichteinstrahlung (bspw. spiegelnd / schattierend / ...) festgelegt werden.

Mit diesen Werkzeugen können sehr detaillierte Modellgenerierungen erfolgen, die nahezu realitätsgerecht sind.

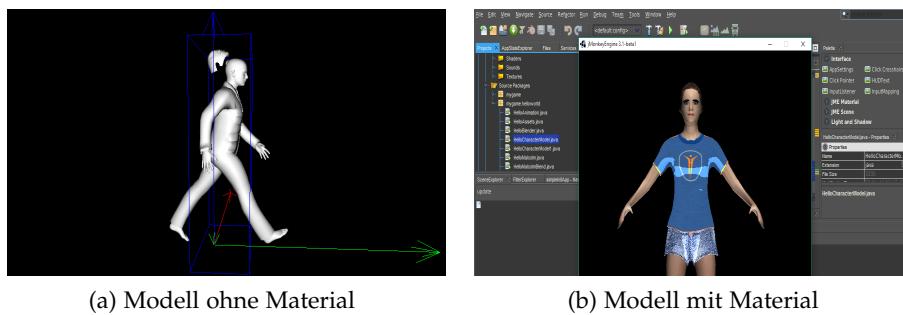


Abbildung 2: Modell mit und ohne Material [3D14]

2.3.5 User-Input

Aus einem 3-dimensionalen Gebilde wird erst dann ein Spiel, wenn Interaktion mit dem Spieler stattfindet. Dazu müssen Tasteneingaben, Mauseingaben oder gegebenenfalls auch Toucheingaben abgefangen und verarbeitet werden. Hierbei werden die aus der ProkSy-Vorlesung bekannten Listener-Klassen verwendet. Bezogen auf unser Spiel wurden folgende Aktionen durch Listener abgefangen:

1. W/A/S/D: Dient zur Bewegung innerhalb des Spielraumes.
2. Pfeiltasten/Maus: Zur Rundumsicht entsprechend einer Kopf-Bewegung.
3. Taste "B": Aufnahme eines gefundenen Buches.
4. Taste "L": Ein-/Ausschalten der Taschenlampe.
5. Leertaste: Sprung

Zur Realisierung stehen in jme3 zwei wichtige Listener-Klassen zur Verfügung: Der *ActionListener* und der *AnalogListener*. Ersterer sollte verwendet werden, wenn einzelne Aktionen erfolgen wie z.B. einmaliges Drücken der Taste B, zum Aufsammeln eines Buches.

Die zweite Klasse wiederum, falls dauerhafte Events wie Gedrückthalten von W zur Vorwärtsbewegung abgefangen werden sollen.

Im Programmcode müssen dann entsprechende Aktionen erfolgen, damit der Input auch eine Auswirkung auf das Spiel hat. Beim Drücken von W müsste etwa der Richtungsvektor sowie die aktuelle Position abgefangen werden um eine Vorwärtsbewegung um $x = walkingSpeed$ Richtungseinheiten zu erzeugen.

2.3.6 Kollisionserkennung

Damit man als Charakter nicht durch die gesamte Spielwelt gehen kann, müssen entsprechende Kollisionserkennungen eingebaut werden. So sollte der Spieler beispielsweise stoppen, wenn er sich in einen Baum hinein bewegen würde. Dazu kann in der Engine der folgende Code realisiert werden:

```
CollisionShape shape = CollisionShapeFactory.createMeshShape(treeShape);
baumControl = new RigidBodyControl(shape);
baum.addControl(baumControl);
```

Im ersten Schritt wird eine Form erstellt, welche das Modell annähert oder teilweise sogar mit ihm übereinstimmt. Bei Bäumen reichen theoretisch auch einfache Zylinder, da man mit der Baumkrone im Spiel sowieso nicht in Kontakt kommt. Im obigen Beispiel wird allerdings auf das Mesh, d.h die tatsächliche Form zurückgegriffen. Danach wird ein Art 'Überwacher' instantiiert, welcher sich um die Kollisionsfunktionalität kümmert. Dieser "kontrolliert" wann sich Modelle überlappen und verhindert dadurch ein Bewegen durch diese. Optional kann als Parameter auch die physikalische Masse des Objekts übergeben werden.

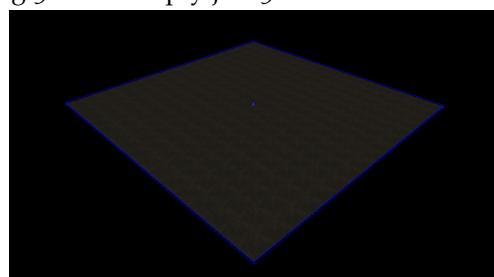
Im letzten Schritt wird dieser Control zum gewünschten Objekt hinzugefügt, damit eine entsprechende Kollisionserkennung stattfinden kann.

Bei Erzeugung einer Spielumgebung im *SceneComposer* werden die physikalischen Eigenschaften bereits implizit implementiert.

2.3.7 Erzeugung einer Spielumgebung

Um die Welt in der wir uns im Spiel befinden vorher genau zu definieren, kann man in der jMonkey SDK eine eigene Scene erstellen. Hat man eine neue "Empty jME3 Scene" erstellt, so erscheint der SceneComposer, in welchem man die Scene betrachten kann. Nun sollte man der Scene vorerst ein Terrain hinzufügen. Ein Terrain ist ein Gelände, welches man nach seinen Vorzügen gestalten kann. Somit sind Gebirge und Hügel vorstellbar. Man kann dieses Terrain im sogenannten *Terrain Editor* bearbeiten und nach seinen Wünschen anpassen.

Abbildung 3: New Empty jME3 Scene mit erstelltem Terrain

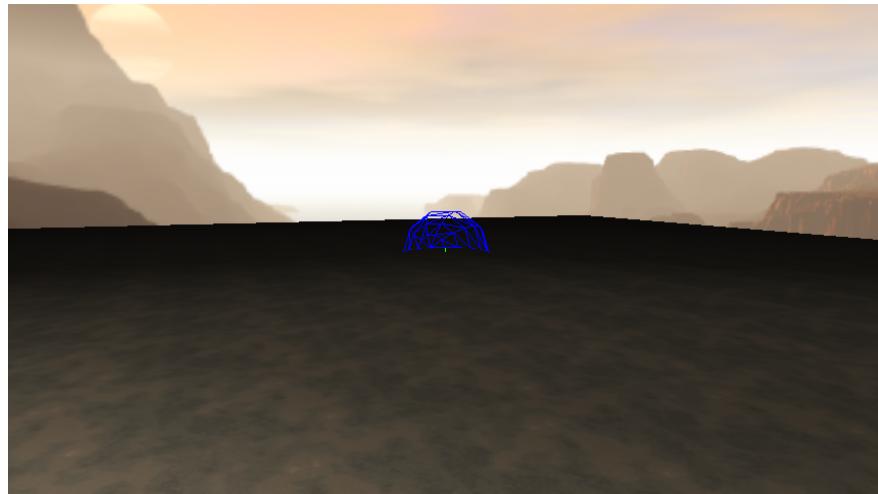


Nun kann man der Spielumgebung noch einige Modelle hinzufügen, sodass eine Welt entsteht, in der man sich während des Spiels bewegen kann. Modelle bekommt man relativ schnell und einfach über das Internet, wobei diese meistens nicht für die jMonkeyEngine konstruiert sind, sodass man diese noch konvertieren muss. Man kann diese 3D-Modelle in einem 3D-Tool wie Blender importieren und für die jMonkeyEngine exportieren. Dies ist jedoch leichter gesagt als getan, denn viele Modelle verwenden Features, die nicht mit der jMonkeyEngine kompatibel sind. So benutzen viele Designer statt einer Lampe eine Sun als Lichtquelle, oder Modifiers, welche das Modell leicht verändern.

Es war sehr viel Arbeit gute Modelle zu finden, die man in der jMonkeyEngine ohne viel Aufwand benutzen kann. Um ein eigenes Spiel wirklich komplett zu programmieren, empfiehlt es sich daher, dass man sich mit dem Design von 3D-Modellen auseinander setzt, wodurch man diese Mechanismen des Imports besser verstehen kann. Auf dieser Basis kann man sich zusätzlich eigene Modelle erstellen, welche perfekt zu dem eigenen Spiel passen.

Möchte man der Welt neben ein paar Modellen außerdem eine realistische Umgebung geben, so darf man den Himmel und den Horizont nicht vergessen. Auch hier hat die jMonkeyEngine vorgefertigte Libraries und im SceneComposer ist es relativ einfach, einen geeigneten Himmel hinzuzufügen. Nach einer kleinen Suche im Internet findet man schnell passende Texturen, welche gegebenenfalls aber auch selbst erstellt werden können. Ein Himmel, auch *Skybox* genannt, besteht entweder aus sechs verschiedenen Bildern, welche die einzelnen Richtungen (Norden, Süden, Osten, Westen, Oben, Unten) ausmachen, oder aus einem einzelnen, welches diese verschiedenen Bilder zusammenfasst. Hier ein Beispiel wie eine Skybox im SceneComposer mit unserem Terrain aussieht:

Abbildung 4: Terrain mit Skybox



Um in der Scene beim Spielen die geeignete Atmosphäre zu erstellen, ist die Verwendung von Licht unbedingt notwendig. Verzichtet man darauf, kann man nur wenig erkennen. Es gibt dazu im *com.jme3.light* Paket verschiedene Lichttypen, die man verwenden kann:

- PointLight
- DirectionalLight
- AmbientLight
- SpotLight

Grundsätzlich kann man jedem Licht eine Farbe zuweisen, um der Welt eine bestimmte Atmosphäre zu verleihen. Nachdem man das Licht nach seinen Wünschen erstellt hat, muss es nun der rootNode mit der addLight Methode bekannt gemacht werden. Ein PointLight scheint von einem Punkt aus in alle Richtungen, und nimmt mit der Entfernung zu dem Lichtpunkt an Intensität ab, hier ein kleines Beispiel:

```
PointLight p = new PointLight();
p.setRadius(20f);
p.setColor(ColorRGBA.Yellow);
p.setPosition(Vector3f.ZERO);
rootNode.addLight(p);
```

Abbildung 5: PointLight



Das PointLight kann sehr gut für Lampen oder sonstigen Lichtquellen verwendet werden, bei denen ein Licht in alle Richtungen ausstrahlt. Im Gegensatz dazu gibt es das DirectionalLight, welches nur in eine Richtung ein Licht ausstrahlt. Dieses wird in der Regel nur einmal in einer Scene verwendet, da es keinen Radius und keine Position hat, es scheint überall in eine Richtung. Hier ein kleines Codebeispiel:

```
DirectionalLight d = new DirectionalLight();
d.setColor(ColorRGBA.White);
d.setDirection(new Vector3f(-.5f, -.5f, -.5f).normalizeLocal());
rootNode.addLight(d);
```

Abbildung 6: DirectionalLight



Möchte man jedoch, dass in der gesamten Scene ein Licht erscheint, welches die Umgebung gleich stark beleuchtet, so kann man das AmbientLight verwenden. Dieses hat wie das DirectionalLight weder eine Position, noch eine Richtung, es geht es vielmehr darum, überall in der Scene ein Licht zu erstellen, das eine gewissen Grundstimmung erzeugt. Hier ein kleines Beispiel:

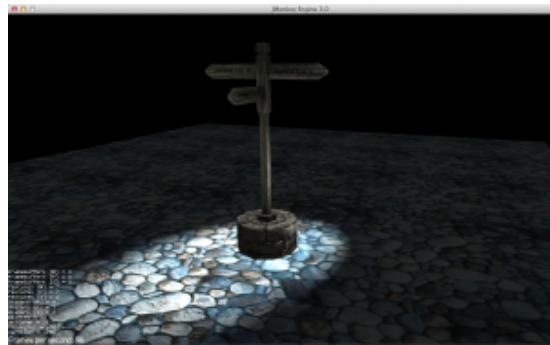
```
AmbientLight al = new AmbientLight();
al.setColor(ColorRGBA.Blue) ;
rootNode.addLight(al);
```

Um der Scene ein Licht hinzuzufügen, welches in eine Richtung scheint und eine Position sowie einen Radius hat, kann man das SpotLight verwenden. Man kann sich dieses Licht ähnlich wie eine Taschenlampe vorstellen, die in eine Richtung stahlt und eine Position hat. Das SpotLight erzeugt einen Lichtkegel in eine bestimmte Richtung mit einer gewissen Intensität. Zusätzlich gibt es noch die Möglichkeit, dem Lichtkegel zwei verschiedene Winkel zu geben: Den inneren Winkel, welcher den hellen Bereich eines Lichtkegels beschreibt und den äußeren Winkel, der den äußeren Kreis des Kegels zeichnet. Hier ein kleines Beispiel:

```
SpotLight s = new SpotLight();
s.setSpotRange(100f);
s.setSpotInnerAngle(5f * FastMath.DEG_TO_RAD);
s.setSpotOuterAngle(20f * FastMath.DEG_TO_RAD);
s.setColor(ColorRGBA.White);
s.setPosition(cam.getLocation());
s.setDirection(cam.getDirection());
rootNode.addLight(s);
```

Im Beispiel auf der nächsten Seite zeigt das Licht von der Kameraposition in die Blickrichtung der Kamera. Hierbei ist es notwendig, diese Position während des Spiels zu aktualisieren (zum Beispiel in der simpleUpdate Methode), sodass die Taschenlampe sich mit dem Spieler bewegt, als hätte er sie in der Hand. Alternativ dazu kann man einen *LightControl* verwenden, der dafür sorgt, dass ein SpotLight oder PointLight einem Spatial zugehörig ist, sodass es der Position des Spatials folgt.

Abbildung 7: SpotLight



Hier sind auch weitere ähnliche Einsatzmöglichkeiten denkbar, zum Beispiel die Scheinwerfer eines Autos.

```
PointLight p = new PointLight();
p.setRadius(20f);
p.setColor(ColorRGBA.Yellow);
p.setPosition(Vector3f.ZERO);
rootNode.addLight(p);
LightControl lightControl = new LightControl(p);
spatial.addControl(lightControl);
```

2.3.8 Hinzufügen von Audio

Bei großen rollen-basierten Spielen erzeugen die Sound-Effekte, neben den grafischen Eigenschaften, den Hauptbestandteil der entsprechenden Atmosphäre. In unserem Spiel wurden folgende Sound-Effekte verwendet:

1. Horror-Theme Soundtrack als Hintergrundmusik
2. Donner-und Regensounds, welche zufällig abgespielt werden
3. Fußstapfen im Wald
4. Spezial Effekte wie beispielsweise: Knisterndes Feuer, Wolfs-Heulen, Spieler-Atmen, Herzklopfen...

Um Audio dem Spiel hinzuzufügen sind beispielhaft die folgenden Code-Zeilen notwendig (bei bereits existierender Instanz-Variable).

```
audio_nature = new AudioNode(assetManager, "Sound/nature.mp3", true); // Laden
audio_nature.setLooping(true); // Aktiviere wiederholendes Abspielen
audio_nature.setPositional(true); // 3D-Audio-Effekte
audio_nature.setVolume(3); // Volume festlegen
rootNode.attachChild(audio_nature);
audio_nature.play(); // Abspielen des Sounds
```

2.3.9 Physikalische Modellierung

Die jme3 Engine erlaubt durch ihre Physik-Engine Objekten verschiedenes physikalisches Verhalten zuzuordnen. Die Kräfte, welche wirken sind dann je nach Masse verschieden.

In unserem Spiel haben wir hauptsächlich von der Schwerkraft Gebrauch gemacht, welche sich durch den Befehl `setGravity()` auf Spatials anwenden lässt.

2.3.10 Effekte und Details

Um dem Spiel mehr Leben einzuhauchen können verschiedenste Spezialeffekte erstellt werden. Im Folgenden wird auf zwei wichtige Beispiele eingegangen und wie diese in der jMonkeyEngine umgesetzt werden können.

2.3.10.1 Nebel

In unserem Wald haben wir zur besseren Atmosphäre einen Nebeleffekt hinzugefügt. Genaugenommen ist dies lediglich eine Erhellung der Pixel in Abhängigkeit des Abstandes vom Spieler. Es gibt zwei Möglichkeiten im Code Nebel zu erzeugen:

1. Programmieren eines Shaders
2. Verwendung des FogFilters in jme3

Da wir uns für die Theorie dahinter interessierten haben wir uns über das allgemeine Vorgehen für 1. informiert aber letztendlich den FogFilter verwendet. Die Berechnung des Nebels erfolgt exponentiell mit Berücksichtigung von Lichteffekten, abhängig vom Abstand zum Spieler. Entsprechende Erklärungen können unter [Cr14] nachgeschlagen werden.

$$\text{FinalFogColor} = (1.0 - e^{d \cdot b_1}) * \text{fogColor} + e^{d \cdot b_2} * \text{lightColor} \quad (1)$$

d = Abstand zum Spieler

b = Nebeldichte je nach Lichteffekt

Abbildung 8: Exponentieller Nebel - Ergebnis



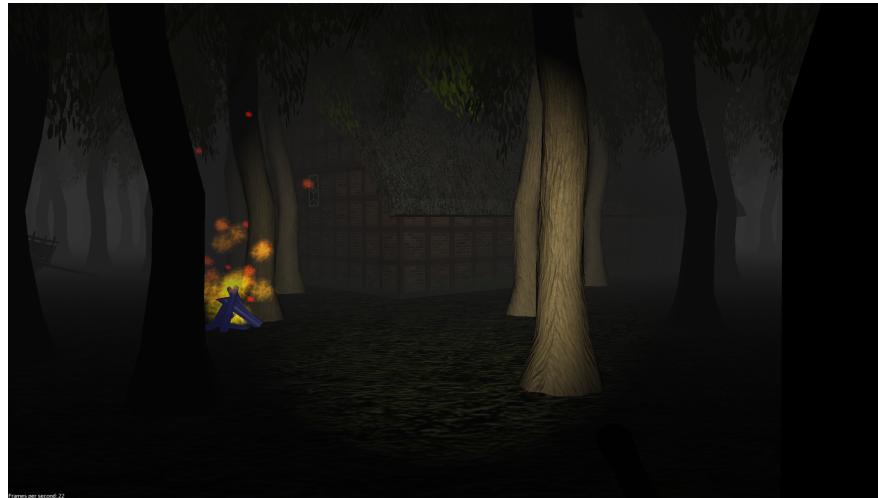
2.3.10.2 Partikeleffekte

In fast jeder 3D-Game-Engine gibt es eine Unterstützung von Partikeleffekten. Dabei werden Partikel verschiedenster Formen und Größen mit unterschiedlichen Geschwindigkeiten in den Raum gezeichnet. Hiermit lassen sich beispielsweise Effekte wie ein brennendes Feuer, Schnee, Regen oder Explosionen erstellen. In jme3 existiert hierfür die Klasse *ParticleEmitter*, welche sich um das Verhalten der Partikel kümmert. In unserem Progman-Spiel kam diese in zwei Fällen zur Anwendung: Leichter Regen, sowie ein kleines Feuerchen in der Nähe eines Hauses.

Im Folgenden ein Code-Beispiel wie Partikeleffekte erzeugt werden können, sowie eine Abbildung des Ergebnisses.

```
ParticleEmitter pm = new ParticleEmitter("effect", Type.Triangle, 60);
Material pmMat = new Material(assetManager,
    "Common/MatDefs/Misc/Particle.j3md");
pmMat.setTexture("Texture", assetManager.loadTexture("Effects/rain.png"));
pm.setMaterial(pmMat);
pm.setImagesX(1);
pm.setImagesY(1);
rootNode.attachChild(pm);
```

Abbildung 9: Partikeleffekt - Ergebnis



2.4 OPTIMIERUNG DES PROGRAMMS

Wenn wie oben beschrieben einige Modelle und Effekte in die Spielumgebung eingebunden werden, können der Prozessor bzw. die Grafikkarte sehr schnell an ihre Grenzen stoßen. Dabei spielt die Anzahl der *Vertexes* bzw. *Triangles* eine zentrale Rolle. Jedes Modell hat unter Umständen einige tausend Dreiecke, sodass sich dies in einem Spiel sehr leicht aufsummieren kann.

In unserem Spiel gibt es zum Beispiel knapp 2000 Bäume, welche alle gerendert werden müssen: Vereinfacht man dort das Modell des Baumes, hat dies viel Potenzial, das gesamte Spiel zu beschleunigen. Mit Hilfe von *F5* kann man in jMonkey während des Spiels anzeigen lassen, wie viele *Triangles* und *Vertexes* gerade zu rendern sind. Es versteht sich von selbst, dass ein Spiel mit einigen Millionen *Vertexes* viel zu aufwendig wird, weshalb die Framerate meist auf nahezu null sinkt. Um dies zu verhindern, muss man also die Gesamtanzahl an *Triangles* und *Vertexes* verringern. Dies ist grundsätzlich durch die Minimierung der Anzahl von Modellen oder durch die Minimierung der Anzahl an *Triangles* und *Vertexes* innerhalb eines Modells möglich.

Diese haben dafür oft ein sogenanntes *Level Of Detail* (kurz LOD), welches je nach Level ein Modell genauer zeichnet. Hier könnte man implementieren, dass das Modell genauer gezeichnet wird, wenn die Distanz zu dem Modell klein ist, sodass die naheliegenden Modelle detailliert, die weit entfernten Modelle jedoch nur grob gezeichnet werden.

Interessanterweise fiel uns auf, dass das Terrain selbst (also der Untergrund des Spielers) sehr viele *Triangles* besitzt. Das liegt daran, dass das Terrain dafür ausgelegt ist, aufwendige Umgebungen darzustellen (sog. *heightmaps*). So können Gebirge oder sonstige Unebenheiten sehr fein erstellt werden. Der Nachteil dabei ist jedoch, dass es sehr aufwendig wird das Terrain selbst ohne Modelle zu rendern, obwohl dieses wie in unserem Spiel einfach nur eben sein kann. Es ist möglich die Feinheit des Terrains anzupassen, wordurch schnell viele tausend Dreiecke gespart werden können. Deshalb ist es unbedingt notwendig bei der Programmierung eines 3D-Spiels auf die Framerate und Komplexität der Welt zu achten, um das Spiel flüssig zu gestalten.

2.4.0.1 Minimierung der Anzahl von Modellen

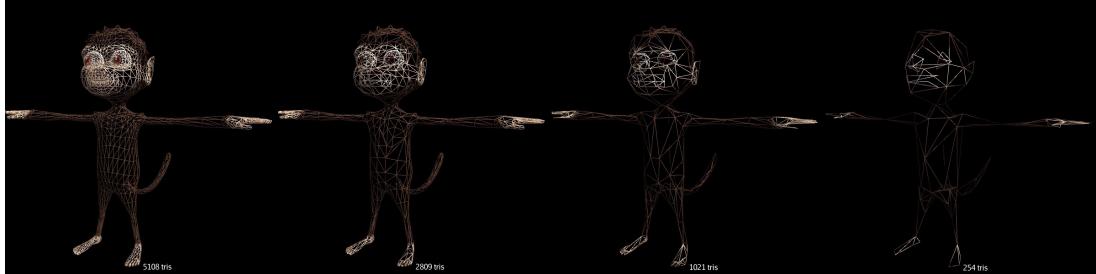
Um eine hohe Spielgeschwindigkeit zu gewährleisten bietet sich vorerst an, das Spiel möglichst simpel zu halten. Dies kann zum Beispiel erreicht werden, indem die Anzahl der benutzten Modelle reduziert wird. Also sollte man keine unnötigen Modelle verwenden, die nicht gebraucht werden. Man könnte zum Beispiel Modelle, welche von der Kameraposition im Spiel weit entfernt sind ausblenden, da diese sowieso nicht gesehen werden (Stichwort Culling). In jMonkey kann man die Distanz einstellen, ab welcher die Modelle nicht mehr gerendert werden. Hier ein Beispiel aus unserem Code:

```
camera.setFrustumPerspective(45f, (float)cam.getWidth() /  
    cam.getHeight(), 1f, 100f); // bis 100 Meter
```

Somit muss in unserem Beispiel nicht jeder der fast 2000 Bäume gerendert werden, sondern nur in der Nähe und im Sichtbereich des Spielers. Auf diesem Weg kann sehr einfach die Komplexität der darzustellenden Welt reduziert werden.

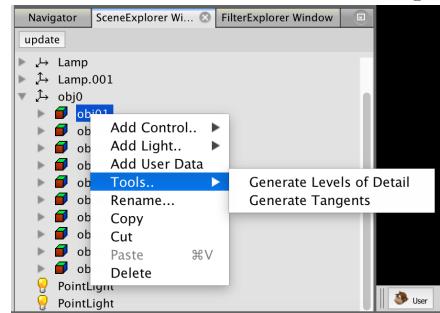
2.4.0.2 Level of Detail (LOD)

Abbildung 10: Die jMonkey Figur in verschiedenen LODs



In der jMonkeyEngine ist es grundsätzlich vorgesehen, dass man seinen Modellen eigene LODs gibt. Damit kann man anhand dieser LODs die Komplexität des Modells im Laufe des Spiels bzw. vor dem Spiel verändern, indem man die entsprechenden Werte der Levels anpasst. Dabei gilt es noch zu beachten, dass die Modelle selbst keine LODs haben, sondern nur die Geometries, welchen sie zugrunde liegen. Man kann innerhalb des SceneExplorers die einzelnen Geometries in den Modellen auswählen, und so eigene LODs generieren.

Abbildung 11: LODs mit der SDK im SceneExplorer erstellen



Ein alternativer Weg ist, dass man der Geometry eines vorhandenen Spatial während der Laufzeit über den Java-Code neue LODs zuweist. Dies hat den Vorteil, dass man die LODs dynamisch erstellen kann. Ein anderer Grund kann sein, dass man das Modell, das man benutzen will, nicht bearbeiten kann. Bei den Bäumen die wir in unseren Spiel benutzt haben kann man zum Beispiel über den SceneComposer keine LODs hinzufügen, da das Modell des Baums nur in der Testdata Library von jMonkey enthalten ist.

Um LODs zu erstellen muss zunächst für jede Geometry ein LodGenerator erstellt werden. Danach wird die *bakeLods*-Methode aufgerufen, welcher man die Reduction-Method und den reductionValue übergeben muss. Es gibt drei verschiedene Möglichkeiten für die *reductionMethod*: Am häufigsten verwendet wird die PROPORTIONAL Methode, welche die Polygone um einen gewissen Prozentsatz reduziert.

Die COLLAPSE_COST Methode reduziert die Anzahl der Ecken solange, bis die Reduktionskosten den übergebenen Wert erreichen. Die CONSTANT Methode entfernt die gegebene Anzahl an Polygonen.

```
LodGenerator lod = new LodGenerator(geometry);
lod.bakeLods(reductionMethod, reductionValue);
```

Der Methode `bakeLods` kann man auch mehr als nur einen `reductionValue` übergeben und somit mehrere LODs erstellen. Die LODs sind gedacht, um das Spiel zu vereinfachen, nicht jedoch, um die Spielwelt zu verunstalten. Ein stark vereinfachtes Modell kann dem Spieler durchaus negativ auffallen. Deshalb ist es sinnvoll, das Level of Detail während des Spiels automatisch anzupassen.

Dafür gibt es die vorgefertigte `LodControl`-Klasse, welche sicherstellt, dass das Level of Detail ausgehend von der Distanz zwischen Kamera und dem Modell angemessen gesetzt wird, hier ein Beispiel:

```
LodControl lc = new LodControl();
lc.setTrisPerPixel(trisPerPixel);
myPrettyGeo.addControl(lc);
rootNode.attachChild(myPrettyGeo);
```

2.5 KRITIK AN DER JMONKEYENGINE3 UND FAZIT

Wir haben die jMonkeyEngine ausgewählt, weil sie komplett Java basiert ist, und weil sie eine gute SDK hat, in der man sehr einfach und schnell sein erstes 3D Spiel erstellen kann. Die Funktionen, die die jMonkeyEngine hat sind vielfältig. Wir möchten in diesem Abschnitt nun darauf eingehen, was wir für Erfahrungen mit der jMonkeyEngine gemacht haben.

Die Grundlagen eines Spiels zu legen fiel uns leicht, denn dazu konnte man die Tutorials gut zu Rate ziehen, jedoch sind wir mit der Programmierung innerhalb der jMonkeyEngine schnell an die Grenzen unseres Wissens gestoßen, und haben auch im Internet wenig hilfreiche Beiträge gefunden. Schenkt man jmonkeyengine.org Glauben, so hat die jMonkeyEngine eine sehr gute Community und viele Probleme sind bereits geklärt worden. Doch als wir beim Programmieren einige Probleme hatten, war eben dies **nicht** gegeben. Oft findet sich ein Foreneintrag zu einem ähnlichen Thema. Liest man diesen Eintrag nun durch, versteht man als Anfänger recht wenig, da sich dort häufig Experten und Core Developers unterhalten. Dazu kommt, dass sehr viele Foreneinträge bereits einige Jahre alt sind und damit auch deren Links und Verweise auf andere Foreneinträge nicht mehr existieren.

Insgesamt hatten wir den Eindruck, als sei die Community vor einigen Jahren aktiv gewesen, jedoch nicht mehr heute. Sucht man bei Google das Wort "jMonkey", so findet man 113.000 Ergebnisse, sucht man stattdessen andere Engines wie "unity engine", so findet man 15.100.000, oder "cry engine" 11.600.000. Neben der nicht überzeugenden Dokumentation gibt es außerdem einige Fehler innerhalb von jMonkey oder der SDK. Das Erstellen eigener LODs hat zum Beispiel nur bei einem von uns beiden funktioniert.

Ein anderer Fall war, dass beim Arbeiten mit einer großen Scene, die Datei plötzlich nicht mehr im SceneComposer geöffnet werden konnte.

Darüber hinaus gibt es noch diverse Fehler in den jMonkeyLibraries: In der SpotLight Klasse aus dem Paket *com.jme3.light* erscheint bei jedem run ein überflüssiger Konsolenaufdruck. Dort wird im Konstruktor ein berechneter Winkel in der Konsole ausgegeben. Laut Internet hatte ein Entwickler vergessen diesen aus dem Code zu nehmen, weshalb er dort bis heute ist.

Ein weiterer großer Kritikpunkt ist die Konvertierung von Modellen in jMonkey3. Beim Verwenden öffentlicher Modelle kann es sehr schnell zu Problemen führen, die auch oft nicht weiter spezifiziert werden. Beim Transformieren und Exportieren von 3D Modellen in jMonkey kam es beispielsweise oft zu Problemen, da keine Umwandlung in den *jme3*-Datentyp möglich war.

Zusammengefasst kann man sagen, dass die jMonkeyEngine eine sehr gute Engine ist, die viele Funktionen beinhaltet und dazu sehr variabel anpassbar ist, jedoch auch ihre Schwächen und längst nicht die riesige Community hat, wie es auf der Website angeben ist. Als Anfänger in der Spieleprogrammierung ist die Game-Engine an sich zu empfehlen, da die Grundfunktionalitäten vermittelt werden. Falls allerdings die Möglichkeit zu anderen Programmiersprachen steht, so sollte man diese nutzen, da Game-Engines mit beispielsweise C++ deutlich ausgereifter und verbreiteter sind.

LITERATURVERZEICHNIS

- [3D14] 3D Model Figures, 2014. Abbildungen:
<https://hub.jmonkeyengine.org/t/changes-to-animations-loading-in-blender-importer-important-for-importer-users/28304/10>.
- [Ba] Basisfunktionalitäten von Spiel-Engines.
- [Cr14] Create a fog shader. <http://in2gpu.com/2014/07/22/create-fog-shader/>.
- [De] Definition von Frustrum Culling nach DelphiGL.
- [Eb] Eberly, David H.: 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics.
- [jMa] jMonkey Engine Funktionalitäten.
- [jMb] jMonkeyEngine Beginners Guide.
- [OB] OBJ von Wavefront Technologies.
- [Th] ThinMatrix: , 3D Game Example.
- [Ue] Uebersicht verschiedener Game-Engines.

ERKLÄRUNG

Hiermit versichere ich, dass ich die vorgelegte Arbeit in allen Teilen selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln einschließlich des Internets und anderer elektronischer Quellen angefertigt habe. Alle Stellen der Arbeit, die ich anderen Werken dem Wortlaut oder dem Sinne nach entnommen habe, sind kenntlich gemacht.

Karlsruhe, 26. Januar 2017

Julian Wadephul und Florian Rottach